CSE 332 Spring 2026
Lecture 24: Concurrency 3 – Deadlock and Wisdom

Nathan Brunelle

http://www.cs.uw.edu/332

Deadlock

- Occurs when two or more threads are mutually blocking each other
- T1 is blocked by T2, which is blocked by T3, …, Tn is blocked by T1
  - A cycle of blocking

- Three requirements for deadlock:
  - Multiple threads each need to acquire **multiple locks**
  - The locks need to be **held at the same time** by the threads
  - The locks may be **acquired in multiple orders**

Analog Example – Need the Trident to Write, Need the Dinosaur to Speak

Nathan

Acquire the Trident

Write your first name in the box

Acquire the Dinosaur

Write your last name in the box while calling out each letter

Release the Dinosaur

Release the Trident

Volunteer

Acquire the Dinosaur

say your first name

Acquire the Trident

Write your name in the box while calling out each letter

Release the Trident

Release the Dinosaur

Bank Account

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        synchronized void transferTo(int amt, BankAccount a) {
                this.withdraw(amt);
               a.deposit(amt);
        }
}
```

Deadlock Example – Locking Order

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

acquire lock for account x b/c transferTo is synchronized **acquire lock for account y** b/c deposit is synchronized **release lock for account y** after deposit **release lock for account x** at end of transferTo	**acquire lock for account y** b/c transferTo is synchronized **acquire lock for account x** b/c deposit is synchronized **release lock for account x** after deposit **release lock for account y** at end of transferTo

Deadlock Example – Deadlock Interleaving

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
y.transferTo(1,x);
```

acquire lock for account x b/c transferTo is synchronized	**acquire lock for account y** b/c transferTo is synchronized
acquire lock for account y b/c deposit is synchronized	**acquire lock for account x** b/c deposit is synchronized
release lock for account y after depost	**release lock for account x** after deposit
release lock for account x at end of transferTo	**release lock for account y** at end of transferTo

Resolving Deadlocks

- Option 1: Address the “multiple locks” requirement
  - Have a coarser lock granularity
  - E.g. one lock for ALL bank accounts
- Option 2: Address the “held at the same time” requirement
  - Have a finer critical section so that only one lock is needed at a time
  - E.g. instead of a synchronized transferTo, have the withdraw and deposit steps locked separately
- Option 3: Address the “acquired in multiple orders” requirement
  - Force the threads to always acquire the locks in the same order
  - E.g. make transferTo acquire both locks before doing either the withdraw or deposit, make sure both threads agree on the order to aquire

Option 1: Coarser Locking

```
static final Object BANK = new Object();
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                synchronized(BANK){
                        this.withdraw(amt);
                        a.deposit(amt);
                }
        }
}
```

Option 2: Finer Critical Section

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                synchronized(this){
                        this.withdraw(amt);
                }
                synchronized(a){
                        a.deposit(amt);
                }
        }
}
```

Option 3: First Get All Locks In A Fixed Order

```
class BankAccount {
        …
        synchronized void withdraw(int amt) {…}
        synchronized void deposit(int amt) {…}
        void transferTo(int amt, BankAccount a) {
                if (this.acctNum < a.acctNum){
                        synchronized(this){
                                synchronized(a){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
                else {
                        synchronized(a){
                                synchronized(this){
                                        this.withdraw(amt);
                                        a.deposit(amt);
                } } }
        }
}
```

Option 3 Requirements

- To acquire the locks in a fixed order you need ALL of:
  - A field that all locks share and that is comparable
  - This field must be unique across all locks
  - The field must not be able to change (must be immutable)

Option 3: Non-Unique

Suppose x.acctNum == y.accNum

Thread 1:

```
x.transferTo(1,y);
```

```
if(this.acctNum < a.acctNum){ // this condition is false
   synchronized(this){
      synchronized(a){
         this.withdraw(amt);
         a.deposit(amt);
} } }
else{ // so we acquire the locks in this order
   synchronized(a){
      synchronized(this){
         this.withdraw(amt);
         a.deposit(amt);
} } }

// y acquired first, x acquired second
```

Thread 2:

```
y.transferTo(1,x);
```

```
if(this.acctNum < a.acctNum){ // this condition is false
   synchronized(this){
      synchronized(a){
         this.withdraw(amt);
         a.deposit(amt);
} } }
else{ // so we acquire the locks in this order
   synchronized(a){
      synchronized(this){
         this.withdraw(amt);
         a.deposit(amt);
} } }

// x acquired first, y acquired second
```

Option 3: Mutable

Suppose x.acctNum < y.accNum

Thread 1:

```
x.transferTo(1,y);
```

Thread 2:

```
temp = y.acctNum;
y.acctNum = x.acctNum;
x.acctNum = temp;
y.transferTo(1,x);
```

```
if(this.acctNum < a.acctNum){




synchronized(this){



   synchronized(a){
        … } } }
else{…}
// x acquired first, y acquired second
```

```
temp = y.acctNum;
y.acctNum = x.acctNum;
x.acctNum = temp; // now x.acctNum > y.accNum
if(this.acctNum < a.acctNum){
   synchronized(this){

      synchronized(a){
         … } } }
else{…}



// y acquired first, x acquired second
```

Parallel Code Conventional Wisdom

Memory Categories

All memory must fit one of three categories:

1. Thread Local: Each thread has its own copy
2. Shared and Immutable: There is just one copy, but nothing will ever write to it
3. Shared and Mutable: There is just one copy, it may change
   - Requires Synchronization!

Thread Local Memory

- Whenever possible, avoid sharing resources
- Dodges all race conditions, since no other threads can touch it!
  - No synchronization necessary! (Remember Ahmdal's law)
- Use whenever threads do not need to communicate using the resource
  - E.g., each thread should have its own Random object
- In most cases, most objects should be in this category

Immutable Objects

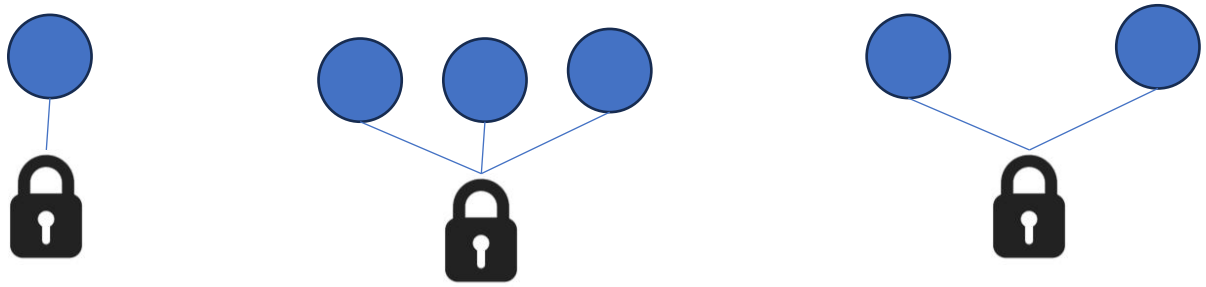
- Whenever possible, avoid changing objects
  - If you need to change an object, instead make a new object with the modifications
- Parallel reads are not data races
  - If an object is never written to, no synchronization necessary!
- Many programmers over-use mutation, minimizing it improves parallelism

Shared and Mutable Objects

- For everything else, use locks
- Avoid all data races
  - Every read and write should be projected with a lock, even if it “seems safe”
  - Almost every Java/C program with a data race is wrong
- Even without data races, it still may be incorrect
  - Watch for bad interleavings as well!

Consistent Locking

- For each location needing synchronization, have a lock that is always held when reading or writing the location
- The same lock can (and often should) “guard” multiple fields/objects
  - Clearly document what each lock guards!
  - In Java, the lock should usually be the object itself (i.e. “this”)
- Have a mapping between memory locations and lock objects and stick to it!



Lock Granularity

- Coarse Grained: Fewer locks guarding more things each
  - One lock for an entire data structure
  - One lock shared by multiple objects (e.g. one lock for all bank accounts)
- Fine Grained: More locks guarding fewer things each
  - One lock per data structure location (e.g. array index)
  - One lock per object or per field in one object (e.g. one lock for each account)
- Note: there’s really a continuum between them…

Example: Separate Chaining Hashtable

- Coarse-grained: One lock for the entire hashtable
- Fine-grained: One lock for each bucket
- Which supports more parallelism in insert and find?
- Which makes rehashing easier?
- What happens if you want to have a size field?

Lock Granularity Tradeoffs

- Coarse-Grained Locking:
  - Simpler to implement and avoid race conditions
  - Faster/easier to implement operations that access multiple locations (because all guarded by the same lock)
  - Much easier for operations that modify data-structure shape
- Fine-Grained Locking:
  - More simultaneous access (performance when coarse grained would lead to unnecessary blocking)
  - Can make multi-location operations more difficult: say, rotations in an AVL tree
- Guideline:
  - Start with coarse-grained, make finer only as necessary to improve performance

Similar But Separate Issue: Critical Section Granularity

- Coarse-grained
  - For every method that needs a lock, put the entire method body in a lock
- Fine-grained
  - Keep the lock only for the sections of code where it's necessary
- Guideline:
  - Try to structure code so that expensive operations (like I/O) can be done outside of your critical section
  - E.g., if you're trying to print all the values in a shared tree, copy items into an array inside your critical section, then print the array's contents outside.
    - Lock the tree when traversing it, the array is thread-local so printing happens in parallel

Atomicity

- Atomic: indivisible
- Atomic operation: one that should be thought of as a single step
- Some sequences of operations should behave as if they are one unit
  - Between two operations you may need to avoid exposing an intermediate state
  - Usually ADT operations should be atomic
    - You don’t want another thread trying to do an insert while another thread is rotating the AVL tree
- Think first in terms of what operations need to be atomic
  - Design critical sections and locking granularity based on these decisions

Use Pre-Tested Code

- Whenever possible, use built-in libraries!
- Other people have already invested tons of effort into making things both efficient and correct, use their work when you can!
  - Especially true for concurrent data structures
  - Use thread-safe data structures when available
    - E.g. Java as ConcurrentHashMap