CSE 332 Autumn 2023
Lecture 26: Wisdom and Deadlock

Nathan Brunelle

http://www.cs.uw.edu/332
class BankAccount {
  private int balance = 0;
  synchronized int getBalance() { return balance; }
  synchronized void setBalance(int x) { balance = x; }
  synchronized void withdraw(int amount) {
    int b = getBalance();
    if (amount > b)
      throw new WithdrawTooLargeException();
    setBalance(b – amount); }
  // other operations like deposit (which would use synchronized)
}
How to fix this?

class Stack {
    private E[] array = (E[])new Object[SIZE];
    private int index = -1;
    synchronized boolean isEmpty() { ... }
    synchronized void push(E val) { ... }
    synchronized E pop() { ... }
    E peek(){
        E ans = pop();
        push(ans);
        return ans;
    }
}
class Stack {
    private E[] array = (E[])new Object[SIZE];
    private int index = -1;
    synchronized boolean isEmpty() { ... }
    synchronized void push(E val) { ... }
    synchronized E pop() { ... }
    synchronized E peek(){
        E ans = pop();
        push(ans);
        return ans;
    }
}

Make a bigger critical section
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

• Guidance: 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

- **Guidance:** Whenever possible, *avoid changing objects*
  - Make new objects instead
- Parallel reads are not *data races*
  - If an object is never written to, no synchronization necessary!
- Many programmers over-use *mutation*, minimize it
• **Guidance:** 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!
  • Use locks whenever there is an incomplete intermediate state!
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”).

• Guidance: 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?
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

• Guidance: 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

• Guidance:
  • 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 tree, maybe copy items into an array inside your critical section, then print the array’s contents outside.
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

• Guidance: 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

• **Guidance: 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
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
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);
    }
}
**The Deadlock**

**Thread 1:**

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

- **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*

**Thread 2:**

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

- **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*

**Expected Behavior:**

Thread 2 items from a stack are popped in LIFO order.
### The Deadlock

#### Thread 1:

x.transferTo(1,y);

- **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

#### Thread 2:

y.transferTo(1,x);

- **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

#### Expected Behavior:

Thread 2 items from a stack are popped in LIFO order.
Resolving Deadlocks

- Deadlocks occur when there are multiple locks necessary to complete a task and different threads may obtain them in a different order.

- **Option 1**: Have a coarser lock granularity.
  - E.g. one lock for ALL bank accounts.

- **Option 2**: 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**: 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 acquire.
Option 1: Coarser Locking

```java
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);
                }
            }
        }
    }
}

Depth-First Search

1. Start from node A, visit B, then visit everything reachable before moving on to C.
Depth-First Search

• Input: a node $s$
• Behavior: Start with node $s$, visit one neighbor of $s$, then all nodes reachable from that neighbor of $s$, then another neighbor of $s$, ...
• Output:
  • Does the graph have a cycle?
  • A topological sort of the graph.
void dfs(graph, s){
    Stack found = new Stack();
    found.pop(s);
    mark s as “visited”;
    While (!found.isEmpty()){
        current = found.pop();
        for (v : neighbors(current)){
            if (!v marked “visited”){
                mark v as “visited”;
                found.push(v);
            }
        }
    }
    Running time: $\Theta(|V| + |E|)$
DFS Recursively (more common)

```java
void dfs(graph, curr){
    mark curr as “visited”;
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;
}
```
Using DFS

- Consider the “visited times” and “done times”
- Edges can be categorized:
  - Tree Edge
    - \((a, b)\) was followed when pushing
    - \((a, b)\) when \(b\) was unvisited when we were at \(a\)
  - Back Edge
    - \((a, b)\) goes to an “ancestor”
    - \(a\) and \(b\) visited but not done when we saw \((a, b)\)
    - \(t_{visited}(b) < t_{visited}(a) < t_{done}(a) < t_{done}(b)\)
  - Forward Edge
    - \((a, b)\) goes to a “descendent”
    - \(b\) was visited and done between when \(a\) was visited and done
    - \(t_{visited}(a) < t_{visited}(b) < t_{done}(b) < t_{done}(a)\)
  - Cross Edge
    - \((a, b)\) goes to a node that doesn’t connect to \(a\)
    - \(b\) was seen and done before \(a\) was ever visited
    - \(t_{done}(b) < t_{visited}(a)\)
boolean hasCycle(graph, curr)
{  
    mark curr as “visited”;  
    cycleFound = false;  
    for (v : neighbors(current))
    {  
        if (v marked “visited” && ! v marked “done”)
        {  
            cycleFound=true;  
        }
        if (! v marked “visited” && !cycleFound)
        {  
            cycleFound = hasCycle(graph, v);
        }
    }
    mark curr as “done”;  
    return cycleFound;
}
Topological Sort

• A Topological Sort of a directed acyclic graph $G = (V, E)$ is a permutation of $V$ such that if $(u, v) \in E$ then $u$ is before $v$ in the permutation.
DFS Recursively

```c
void dfs(graph, curr){
    mark curr as “visited”;  
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”; 
}
```
DFS: Topological sort

def dfs(graph, s):
    seen = [False, False, False, ...] # length matches |V|
    done = [False, False, False, ...] # length matches |V|
    dfs_rec(graph, s, seen, done)

def dfs_rec(graph, curr, seen, done):
    mark curr as seen
    for v in neighbors(curr):
        if v not seen:
            dfs_rec(graph, v, seen, done)
    mark curr as done

Idea: List in reverse order by finish time
DFS Recursively

```c
void dfs(graph, curr){
    mark curr as “visited”;
    for (v : neighbors(current)){
        if (! v marked “visited”){
            dfs(graph, v);
        }
    }
    mark curr as “done”;
}
```

Idea: List in reverse order by finish time
DFS: Topological sort

List topSort(graph) {
    List<Nodes> finished = new List<>();
    for (Node v : graph.vertices) {
        if (!v.visited) {
            finishTime(graph, v, finished);
        }
    }
    finished.reverse();
    return finished;
}

void finishTime(graph, curr, finished) {
    curr.visited = true;
    for (Node v : curr.neighbors) {
        if (!v.visited) {
            finishTime(graph, v, finished);
        }
    }
    finished.add(curr)
}

Idea: List in reverse order by finish time

finished: