Memory Sharing With ForkJoin

• Idea of ForkJoin:
  • Reduce span by having many parallel tasks
  • Each task is responsible for its own portion of the input/output
  • If one task needs another’s result, use join() to ensure it uses the final answer

• This does not help when:
  • Memory accessed by threads is overlapping or unpredictable
  • Threads are doing independent tasks using same resources (rather than implementing the same algorithm)
Example: Shared Queue

enqueue(x){
    if ( back == null ){
        back = new Node(x);
        front = back;
    }
    else {
        back.next = new Node(x);
        back = back.next;
    }
}
Concurrent Programming

• Concurrency:
  • Correctly and efficiently managing access to shared resources across multiple possibly-simultaneous tasks

• Requires synchronization to avoid incorrect simultaneous access
  • Use some way of “blocking” other tasks from using a resource when another modifies it or makes decisions based on its state
  • That blocking task will free up the resource when it’s done

• Warning:
  • Because we have no control over when threads are scheduled by the OS, even correct implementations are highly non-deterministic
  • Errors are hard to reproduce, which complicates debugging
Bank Account Example

- The following code implements a bank account object correctly for a synchronized situation
- Assume the initial balance is 150

```java
class BankAccount {
    private int balance = 0;
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // other operations like deposit, etc.
}
```

What Happens here?

```
withdraw(100);
withdraw(75)
```
Bank Account Example - Parallel

• Assume the initial balance is 150

```java
class BankAccount {
    private int balance = 0;
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
    }
    // other operations like deposit, etc.
}
```

Thread 1:
withdraw(100);

Thread 2:
withdraw(75);
Interleaving

• Due to time slicing, a thread can be interrupted at any time
  • Between any two lines of code
  • Within a single line of code
• The sequence that operations occur across two threads is called an interleaving
• Without doing anything else, we have no control over how different threads might be interleaved
A “Good” Interleaving

• Assume the initial balance is 150

Thread 1:
withdraw(100);

Thread 2:
withdraw(75);

int b = getBalance();
if (amount > b)
    throw new Exception();
setBalance(b – amount);

int b = getBalance();
if (amount > b)
    throw new Exception();
setBalance(b – amount);
A “Bad” Interleaving
• Assume the initial balance is 150

Thread 1:
withdraw(100);

Thread 2:
withdraw(75);

int b = getBalance();
if (amount > b)
throw new Exception();
setBalance(b – amount);

int b = getBalance();
if (amount > b)
throw new Exception();
setBalance(b – amount);
A Bad Fix

• Assume the initial balance is 150

```java
class BankAccount {
    private int balance = 0;
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        if (amount > getBalance())
            throw new WithdrawTooLargeException();
        setBalance(getBalance() - amount);
    }
    // other operations like deposit, etc.
}
```
A still “Bad” Interleaving

• Assume the initial balance is 150

Thread 1:
withdraw(100);

Thread 2:
withdraw(75);

if (amount > getBalance())
    throw new Exception();
setBalance(getBalance() – amount);

if (amount > getBalance())
    throw new Exception();
setBalance(getBalance() – amount);
What we want – Mutual Exclusion

• While one thread is withdrawing from the account, we want to exclude all other threads from also withdrawing

• Called mutual exclusion:
  • One thread using a resource (here: a bank account) means another thread must wait
  • We call the area of code that we want to have mutual exclusion (only one thread can be there at a time) a critical section.

• The programmer must implement critical sections!
  • It requires programming language primitives to do correctly
A Bad attempt at Mutual Exclusion

class BankAccount {
    private int balance = 0;
    private Boolean busy = false;
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        // while (busy) { /* wait until not busy */ }
        busy = true;
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        busy = false;
    }
    // other operations like deposit, etc.
}
A still “Bad” Interleaving

• Assume the initial balance is 150

Thread 1:
withdraw(100);

while (busy) { /* wait until not busy */ }
busy = true;
int b = getBalance();
if (amount > b)
    throw new Exception();
setBalance(b – amount);
busy = false;

Thread 2:
withdraw(75);

while (busy) { /* wait until not busy */ }
busy = true;
int b = getBalance();
if (amount > b)
    throw new Exception();
setBalance(b – amount);
busy = false;
Solution

• We need a construct from Java to do this
• One Solution – A Mutual Exclusion Lock (called a Mutex or Lock)
• We define a Lock to be a ADT with operations:
  • New:
    • make a new lock, initially “not held”
  • Acquire:
    • If lock is not held, mark it as “held”
      • These two steps always done together in a way that cannot be interrupted!
    • If lock is held, pause until it is marked as “not held”
  • Release:
    • Mark the lock as “not held”
Almost Correct Bank Account Example

```java
class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        lk.acquire();
        int b = getBalance();
        if (amount > b)
            throw new WithdrawTooLargeException();
        setBalance(b - amount);
        lk.release();
    }
    // other operations like deposit, etc.
}
```

Questions:
1. What is the critical section?
2. What is the Error?
Try...Finally

• Try Block:
  • Body of code that will be run

• Finally Block:
  • Always runs once the program exits try block (whether due to a return, exception, anything!)
Correct (but not Java) Bank Account Example

class BankAccount {
    private int balance = 0;
    private Lock lck = new Lock();
    int getBalance() { return balance; }
    void setBalance(int x) { balance = x; }
    void withdraw(int amount) {
        try{
            lck.acquire();
            int b = getBalance();
            if (amount > b)
                throw new WithdrawTooLargeException();
            setBalance(b - amount);
        } finally {
            lck.release();
        }
    } // other operations like deposit, etc.
}

Questions:
1. Should deposit have its own lock object, or the same one?
2. What about getBalance?
3. What about setBalance?
A still “Bad” Interleaving

- Assume the initial balance is 150

Thread 1:
```java
withdraw(100);
```

Thread 2:
```java
try{
    lk.acquire();
    int b = getBalance();
    if (amount > b)
        throw new Exception();
    setBalance(b - amount);
} finally {
    lk.release();
}
if(getBalance() < 75)
    setBalance(75);
```
What’s wrong here...

class BankAccount {
    private int balance = 0;
    private Lock lk = new Lock();

    int setBalance(int x) {
        try {
            lk.acquire();
            balance = x;
        } finally {
            lk.release();
        }
    }

    void withdraw(int amount) {
        try {
            lk.acquire();
            int b = getBalance();
            if (amount > b)
                throw new WithdrawTooLargeException();
            setBalance(b - amount);
        } finally {
            lk.release();
        }
    }
}

Withdraw calls setBalance!
Withdraw can never finish because in setBalance the lock will always be held!
Re-entrant Lock (Recursive Lock)

• Idea:
  • Once a thread has acquired a lock, future calls to acquire on the same lock will not block progress
  • If the lock used in the previous slide is re-entrant, then it will work!
Re-entrant Lock Details

• A re-entrant lock (a.k.a. recursive lock)
• “Remembers”
  • the thread (if any) that currently holds it
  • a count of “layers” that the thread holds it
• When the lock goes from not-held to held, the count is set to 0
• If (code running in) the current holder calls acquire:
  • it does not block
  • it increments the count
• On release:
  • if the count is > 0, the count is decremented
  • if the count is 0, the lock becomes not-held
Java’s Re-entract Lock Class

- `java.util.concurrent.locks.ReentrantLock`
- Has methods `lock()` and `unlock()`
- Important to guarantee that lock is always released!!
- Recommend something like this:
  ```java
  myLock.lock();
  try {
    // method body
  } finally {
    myLock.unlock();
  }
  ```
How this looks in Java

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

public class BankAccount {
    private int balance = 0;
    private ReentrantLock lck = new ReentrantLock();

    public int setBalance(int x) {
        try {
            lck.lock();
            balance = x;
        } finally {
            lck.unlock();
        }
    }

    public void withdraw(int amount) {
        try {
            lck.lock();
            int b = getBalance();
            if (amount > b) throw new WithdrawTooLargeException();
            setBalance(b - amount);
        } finally {
            lck.unlock();
        }
    }
}
```
Java Synchronized Keyword

• Syntactic sugar for re-entrant locks
• You can use the synchronized statement as an alternative to declaring a ReentrantLock
• Syntax: 
  
  ```
  synchronized( /* expression returning an Object */ ) {statements}
  ```
• Any Object can serve as a “lock”
  • Primitive types (e.g. int) cannot serve as a lock
• Acquires a lock and blocks if necessary
  • Once you get past the “{“, you have the lock
• Released the lock when you pass “}”
  • Even in the cases of returning, exceptions, anything!
  • Impossible to forget to release the lock
Back Account Using Synchronize (Attempt 1)

class BankAccount {

    private int balance = 0;

    private Object lk = new Object();

    int getBalance() {
        synchronized (lk) {
            return balance;
        }
    }

    void setBalance(int x) {
        synchronized (lk) {
            balance = x;
        }
    }

    void withdraw(int amount) {
        synchronized (lk) {
            int b = getBalance();
            if (amount > b) 
                throw new Exception();
            setBalance(b - amount);
        } // deposit would also use synchronized(lk)
    }
}
Back Account Using Synchronize (Attempt 2)

```java
class BankAccount {
    private int balance = 0;
    int getBalance() {
        synchronized (this) { return balance; }
    }
    void setBalance(int x) {
        synchronized (this) { balance = x; }
    }
    void withdraw(int amount) {
        synchronized (this) {
            int b = getBalance();
            if (amount > b)
                throw new Exception();
            setBalance(b – amount);
        }
    } // deposit would also use synchronized(lk)
```

Since we have one lock per account regardless of operation, it’s more intuitive to use the account object itself as the lock!
More Syntactic Sugar!

- Using the object itself as a lock is common enough that Java has convenient syntax for that as well!
- Declaring a method as “synchronized” puts its body into a synchronized block with “this” as the lock
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)
}
Race Condition

• Occurs when the computation result depends on scheduling (how threads are interleaved)
  • We, as programmers can’t influence scheduling of threads
  • We need to write programs that work independent of scheduling

• Data Race:
  • When there is the potential for two threads to be writing a variable in parallel
  • When there is the potential for one thread to be reading a variable while another writes to it

• Bad Interleaving:
  • A race condition other than a data race
  • Usually it looks like exposing a “bad” intermediate state
Example: Shared Stack (no problems so far)

class Stack {
    private E[] array = (E[])new Object[SIZE];
    private int index = -1;
    synchronized boolean isEmpty() {
        return index==-1;
    }
    synchronized void push(E val) {
        array[++index] = val;
    }
    synchronized E pop() {
        if(isEmpty())
            throw new StackEmptyException();
        return array[index--];
    }
}
Race Condition, but no Data Race

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;
    }
}

Critical sections of this code?
Race Condition, including a Data Race

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(){
        System.out.println(index);
        E ans = pop();
        push(ans);
        push(ans);
        return ans;
    }
}
# Peek and `isEmpty`

**Expected Behavior:**
Thread 2 should not see an empty stack if there is a push but no pop.

<table>
<thead>
<tr>
<th>Thread 1:</th>
<th>Thread 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>peek();</code></td>
<td><code>push(x);</code></td>
</tr>
<tr>
<td></td>
<td><code>boolean b = isEmpty();</code></td>
</tr>
<tr>
<td><code>E ans = pop();</code></td>
<td></td>
</tr>
<tr>
<td><code>push(ans);</code></td>
<td></td>
</tr>
<tr>
<td><code>return ans;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>push(x);</code></td>
</tr>
<tr>
<td></td>
<td><code>boolean b = isEmpty();</code></td>
</tr>
</tbody>
</table>
Peek and Push

Expected Behavior:
Thread 2 items from a stack are popped in LIFO order.
Peek and Pop

Thread 1:
peek();

E ans = pop();
push(ans);
return ans;

Thread 2:
push(x);
push(y);
System.out.println(pop());
System.out.println(pop());

push(x);
push(y);
System.out.println(pop());
System.out.println(pop());

Expected Behavior:
Thread 2 items from a stack are popped in LIFO order
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;
    }
}

Make a bigger critical section
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;
    }
}
Did this fix it?

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(){
        return array[index];
    }
}

No! Now it has a data race!
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
  • 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
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?
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 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
• 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
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:

x.transferTo(1,y);

Thread 2:

y.transferTo(1,x);

Expected Behavior:
Thread 2 items from a stack are popped in LIFO order

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
The Deadlock

Thread 1:

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

Thread 2:

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

**Expected Behavior:**
Thread 2 items from a stack are popped in LIFO order.

- **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`
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);
                }
            }
        }
    }
}