Lecture 25: Concurrency

Today

- Reading
  - P&C Section 6
- Objectives
  - Concurrency

Concurrency

- Correctly and efficiently controlling access by multiple threads to shared resources
- Programming model
  - Multiple uncoordinated threads
  - Sharing memory locations
  - Interleaved operations
- Contrast with Divide and Conquer Parallelism
  - Each thread had memory only it accessed
  - Thread blocked until helper threads finished

Concurrency

- Downsides of concurrency
  - behavior depends on order threads access shared resources
  - leads to seemingly non-deterministic behavior
  - hard to replicate bugs
- Benefits of concurrency
  - There are other models of parallelism that require concurrent access
  - Responsiveness – one thread listens for I/O events while another does computation
  - Processor utilization – schedule threads while waiting for I/O completion
  - Failure isolation – One thread fails, others can keep working

+ Example: Bank Account

```
public 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);
    }
    ...
}
```

+ Interleaved operations

- Two threads withdraw $100 from same bank account with balance of $150

time

```
Thread 1                         Thread 2
--------                         --------
int b = getBalance();
                                 int b = getBalance();
                                 if(amount > b)
                                   throw new ...;
                                 setBalance(b - amount);

if(amount > b)
  throw new ...;
setBalance(b - amount);
```

lost a withdrawal!

Mutual Exclusion

- A *critical section* is a piece of code that accesses a shared resource (e.g. the `withdraw` method)
- Mutual exclusion – only allow one thread in the critical section at a time
- Idea: Do Not Disturb
  - Hang a "Do Not Disturb" sign when enter a critical section so other threads know
  - Remove sign when finished
  - Other threads wait until sign is removed

Mutual Exclusion Locks

- Abstract data type that implement mutual exclusion
  - `new` creates a new lock that is not held
  - `acquire` blocks until the lock is not held and then sets the lock to held and returns
  - `release` sets lock to not held
- Locks in Java
  - No explicit `lock` object as in other languages
  - Instead every object is a lock that can be acquired and released
  - Locks are re-entrant, i.e. a thread can re-acquire a lock it already holds

+ Re-entrant Locks in Java

- Re-entrant locks implemented via `synchronized` blocks

```
synchronized (expression) { statements}
```

- `synchronized` acquires the lock (blocks until available)
- `expression` must evaluate to a non-null object
- `statements` execute when lock acquired
- lock is released when execution leaves block *for any reason*

+ Bank Account (Correct code)

```
public 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 WithdrawTooLargeException();
            setBalance(b-amount);
        }
    }
    ...
}
```

re-entrant locks

lock released if exception

+ Bank Account (Better code)

```
public 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 WithdrawTooLargeException();
            setBalance(b-amount);
        }
    }
    ...
}
```

+ Bank Account (Best code)

```
public 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);
    }
    ...
}
```

Race Conditions

- A race condition occurs when the computation result depends on the order that the threads execute (how threads are interleaved)
- Such bugs (by definition) do not exist in sequential programs
- Typically, problem is one thread "sees" invariant-violating intermediate state produced by another thread
- Two types of race conditions
  - Bad interleavings
  - Data races – simultaneous read/write or write/write access to same memory location

Bad Interleaving Example: peek

```
class Stack<E> {
   private E[] array;  // array to hold elements
   private int index; // points to next open slot

   Stack(int size){ array = (E[]) new Object[size]; }

   synchronized boolean isEmpty() {
        return index == 0;
   }
   synchronized void push(E val) {
        if(index == array.length) throw new ...;
        array[index++] = val;
   }
   synchronized E pop() {
        if(index == 0) throw new ...;
        return array[--index];
   }
}
```

Bad Interleaving Example: peek

- Implementing `peek` from a different class
- Forgot to add synchronization!

```
public class C{
   static <E> E myPeekHelperWrong(Stack<E> s) {
        E ans = s.pop();
        s.push(ans);
        return ans;
   }
}
```

Bad Interleaving Example: peek

- `peek` has no *overall* effect on the shared data
  - It is a "reader" not a "writer"
  - Overall result is same stack if no interleaving
- But the way it is implemented creates an inconsistent *intermediate state*
  - Even though calls to `push` and `pop` are synchronized so there are no *data races* on the underlying array
- This intermediate state should not be exposed
  - Leads to several *bad interleavings*

One bad interleaving: peek and push

- Property we want: values are returned from pop in LIFO order
- With peek as written, property can be violated – how?

Thread 1 (peek)

```
E ans = pop();

push(ans);

return ans;
```

Thread 2

```
push(x)

push(y)

E e = pop()
```

Time

The solution

- peek needs synchronization to disallow interleavings
  - The key is to make a *larger critical section*
  - Re-entrant locks allow calls to push and pop
- Just because all changes to state done within synchronized pushes and pops doesn't prevent exposing intermediate state

```
public class C{
   static <E> E myPeekHelperWrong(Stack<E> s) {
        synchronized(s) {
            E ans = s.pop();
            s.push(ans);
            return ans;
        }
   }
}
```

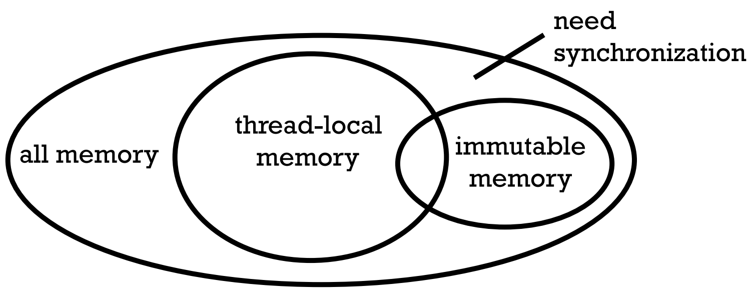
Race conditions

- Examples of data races in the text
- Lesson: *Do not introduce a data race* even if every interleaving you can think of is correct
- Avoiding race conditions on shared resources is difficult
  - Decades of bugs have led to some *conventional wisdom*: general techniques that are known to work

Providing Safe Access

For every memory location (e.g., object field) in your program, you must obey at least one of the following:

1. Thread-local: Only one thread accesses it
2. Immutable: (After initialization) Only read never written
3. Synchronized: Locks used to ensure no race conditions



Extra Slides

Work and Span in terms of DAG

- Recall: T_P = running time if there are P processors available
- Work (T_1): How long it takes to run on 1 processor
  - Corresponds to the number of nodes in the DAG
  - O(N) for simple maps and reductions
- Span (T_∞) : How long it would take with infinite processors
  - Length of the longest path in the DAG
  - Infinite processors must still wait for earlier results to be done
  - *O*(log N) for simple maps and reductions

+ Speed-Up

- Speed-up on P processors is defined as T_1 / T_P
- If speed-up is P as we vary P, we call it perfect linear speed-up
  - Perfect linear speed-up means doubling P halves running time
  - Usually our goal – hard to get in practice
- Parallelism is the maximum possible speed-up: T_1 / T_∞
  - At some point, adding processors won't help
  - Where that point is depends on the span

+ Examples

$$T_P = O((T_1 / P) + T_\infty)$$

- In the algorithms seen so far (e.g., sum an array):
  - $T_1 = O(n)$
  - $T_\infty = O(\log n)$
  - So expect (ignoring overheads): $T_P = O(n/P + \log n)$
- Suppose instead:
  - $T_1 = O(n^2)$
  - $T_\infty = O(n)$
  - So expect (ignoring overheads): $T_P = O(n^2/P + n)$

Amdahl's Law (derive on board)

- Provides upper-bound on speedup given that only part of algorithm can be parallelized
- Amdahl's Law states: $$\frac{T_1}{T_P} = \frac{1}{S + \frac{1-S}{P}}$$
- where S is the percentage of the algorithm that cannot be parallelized
- Corollary of Amdahl's Law: $$\frac{T_1}{T_\infty} = \frac{1}{S}$$